REPORT DOCUMENTATION PAGE

Form Approved
OBM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
June 1998

3. REPORT TYPE AND DATES COVERED
Proceedings

4. TITLE AND SUBTITLE
A Comparison of several Scalable Programming Models

5. FUNDING NUMBERS
Job Order No.
Program Element No. 062435N
Project No.
Task No.
Accession No.

6. AUTHOR(S)
Alan J. Wallcraft

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Research Laboratory
Oceanography Division
Stennis Space Center, MS 39529-5004

8. PERFORMING ORGANIZATION REPORT NUMBER
NRL/PP/7323--98-0019

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Second International Workshop on Software Engineering and Code Design in Parallel Meteorological and Oceanographic Applications, 15-18 June 1998, Scottsdale, AZ

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The NRL Layered Ocean Model (NLOM) is written in the tiled data parallel programming style, and uses an application specific programming interface to isolate operations that require communication. This allows different scalable programming models to be "plugged" into NLOM with relatively little effort. NLOM is similar to other OGCM's, except that it uses a direct Helmholtz's equation solver as part of its semi-implicit time scheme and typically runs with a very large horizontal extent and very few layers in the vertical. There are now several Fortran-based SPMD programming models to chose from on machines with a hardware global memory: a) MPI-1 message passing, b) MPI-2 put/get, c) BSP, d) SHMEM, e) F--, f) OpenMP, and g) HPF. These models are compared and contrasted based on actual experience with NLOM and related kernel benchmarks.

14. SUBJECT TERMS
NLOM (NRL Layered Ocean Model), periodic boundaries, OGCM, Helmholtz's equation, SPMD, and HPF

15. NUMBER OF PAGES
16

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
SAR

GSFC/CP—1998–206860

Second International Workshop on Software Engineering and Code Design in Parallel Meteorological and Oceanographic Applications

Matthew O'Keefe, Christopher Kerr, Editors

Proceedings of a workshop sponsored by the U.S. Department of Energy, Office of Biological and Environmental Research; the Department of Defense, High Performance Computing and Modernization Office; and the NASA Goddard Space Flight Center, Seasonal-to-Interannual Prediction Project, and held at the Camelback Inn, Scottsdale, Arizona June 15–18, 1998

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

June 1998

A Comparison of several Scalable Programming Models

Alan J. Wallcraft

Naval Research Laboratory, Code 7323, Stennis Space Center, MS 39529.
wallcraf@ajax.nrlssc.navy.mil, +1 228 688-4813, Fax: +1 228 688-4759

Abstract

The NRL Layered Ocean Model (NLOM) is written in the tiled data parallel programming style, and uses an application specific programming interface to isolate operations that require communication. This allows different scalable programming models to be "plugged" into NLOM with relatively little effort. NLOM is similar to other OGCM's, except that it uses a direct Helmholtz's equation solver as part of its semi-implicit time scheme and typically runs with a very large horizontal extent and very few layers in the vertical. There are now several Fortran-based SPMD programming models to choose from on machines with a hardware global memory: a) MPI-1 message passing, b) MPI-2 put/get, c) BSP, d) SHMEM, e) F--, f) OpenMP, and g) HPF. These models are compared and contrasted based on actual experience with NLOM and related kernel benchmarks.

Introduction

The NRL Layered Ocean Model, NLOM, has been under continuous development for 20 years [1], [2], [3]. It has been used to model semi-enclosed seas, major ocean basins, and the global ocean. NLOM has been optimized for the problem space of Navy interest, simulation now-casting and prediction of fronts and eddies, and for such problems it is is 10-100 times more efficient (in operations performed per result) than competing OGCM's.

The current implementation of the model uses the tiled data parallel programming style. Consider the following simple serial code fragment:

```
      REAL A(IH+1,JH),DA(IH+1,JH)
      DO J= 1,JH; DO I= 1,IH
        DA(I,J) = DX*(A(I+1,J) - A(I,J))
      ENDDO; ENDDO
```

The arrays A and DA have been extended by a one column "halo" to allow a clean implementation of a periodic boundary. On entry A(IH+1,:) must be identical to A(1,:). The equivalent tiled data parallel version adds a halo on all sides and splits the array into sub-domain tiles:

```
      REAL A(0:IHP+1,0:JHP+1,MP,NP),DA(0:IHP+1,0:JHP+1,MP,NP)
!HPF$ DISTRIBUTE A(*,*,BLOCK,BLOCK),DA(*,*,BLOCK,BLOCK)
      DO N= 1,NP; DO M= 1,MP
        DO J= 1,JHP; DO I= 1,IHP
          DA(I,J,M,N) = DX*(A(I+1,J,M,N) - A(I,J,M,N))
        ENDDO; ENDDO;
      ENDDO; ENDDO;
```

If MP and NP are both 1, this is Single Program Multiple Data (SPMD) domain decomposition. A 2-D, MPE by NPE, grid of processors are all running this identical program, with IHP=IH/MPE and JHP=JH/NPE. Provided the halo is up to date, the code fragment calculates the required values over the subdomain owned by the local processor. Alternatively, if MPxNP represents the number of processors, this is data parallel High Performance Fortran (HPF) [4] and the compiler does not need to generate any off-chip communication. It is also then appropriate for autotasking of the N loop using Fortran 77 compilers on SMP systems.

By using cpp macros, NLOM can select between scalable programming models at compile time while maintaining a single source code. An application specific programing interface (API) is used to isolate operations that require communication (halo updates etcetera). The API must be implemented for each new programming model, but the rest of the code is largely independent of the model used. For more information on scalable NLOM see Wallcraft and Moore [5], [6].

In the area of scalability, NLOM performs similarly to other OGCM's, except that it uses a direct 2-D Helmholtz's equation solver as part of its semi-implicit time scheme and typicaly runs with a very large horizontal extent and very few layers in the vertical. For example, a six layer 1/32 degree Pacific model is typical of "large" problems today and it has a 4096 by 2688 by 6 grid. Since it has so few layers in the vertical, NLOM uses 2-D domain decomposition (with the vertical dimension "on-chip") and performs all operations on 2-D slabs. OGCM's with more degree's of freedom in the vertical might still choose 2-D domain decomposition, but would typically perform communications on an entire 3-D field rather than on individual 2-D slabs. The direct 2-D Helmholtz's equation solver requires transposing from a 2-D to a 1-D domain decomposition, and therefore potentially reduce overall scalability. In general, scalability of NLOM is excellent on current scalable systems (using 64-256 nodes per job) because the 2-D arrays are so large.

SPMD programming models

There are now several Fortran-based SPMD programming models to choose from on machines with a hardware global memory.

MPI-1

Message passing is the most general scheme but it requires the source and target processor to cooperate in the transfer. MPI-1 is the message passing library of choise for SPMD codes, and is available on all platforms [7]. NLOM can use MPI and has cpp macros to hide word length differences and to select between several possible optimizations at compile time: (a) MPI_SENDRECV in place of the default non-blocking point to point calls, (b) SSEND in place of the default SEND, (c) replacements for ALLGATHER, ALLREDUCE(MAX) and ALLREDUCE(MIN) that use a binary tree on one dimension and a ring exchange on the other dimension, and (d) serialized array I/O.

SHMEM

SHMEM is Cray's one-sided put/get direct memory access library [8]. It is only suitable for machines with a hardware global shared memory. SHMEM is available on all Cray and SGI systems (Cray PVP, Cray T3E, SGI Origin 2000), but not on competing SMP or DSM systems from other vendors (e.g. Sun E10000 and HP/Convex SPP-2000). Unlike the other libraries described here, all SHMEM calls are (locally) blocking. Thus the standard Fortran assumption that there is a single thread of control and that any changes to memory or disk (buffers) caused by a subroutine call will happen before it returns is valid for SHMEM, but not necessarily for non-blocking calls in other libraries. The MPI-2 standard [9] has a good discussion of these issues, which can cause optimization problems in Fortran 77 but are much more serious for Fortran 90. SHMEM put updates memory on another processor, but this is not a problem if either (a) put is never used, or (b) the appropriate syncronization calls are included. The typical SHMEM program relies on a fast global barrier, and uses `COMMON` to hold arrays and/or buffers that are accessed from other processors. NLOM can use SHMEM and has cpp macros to hide word length differences and optionally to use local syncronization in place of some global barrier calls.

BSPlib

Bulk Synchronous Parallel delays put/get operations to the end of a "super-step", which allows implementation on machines without a global memory. Note that this implies that the put/get operations are non-blocking. There is a portable implementation, BSPlib, that runs on many machine types [10]. However, BSPlib effectively requires several global barriers at the end of each superstep because it imposes a particular order on puts and gets. There is formally no need for both put's and get's, and NLOM's SHMEM version (for example) never uses put, but there is no way to tell BSPlib to skip put processing. BSPlib has been designed to be called from C, e.g. sizes in bytes and byte offsets. There is a Fortran interface but it is a direct mapping of the C version, and is therefore very obscure to Fortran programmers. However, the library is small enough that it would be relatively easy to build your own (improved) Fortran interface. Unlike SHMEM, BSPlib only allows access to remote memory via pre-registered "windows". This potentially provides a safer interface, and allows non-static arrays to be accessed remotely, but at the cost of more complicated (and slower) code. BSPlib provides an alternative blocking get (on global shared memory machines only) that acts like a SHMEM get, and it is often possible to define a single memory window that includes all named `COMMON` areas. So BSPlib can be made to look almost exactly like SHMEM. However, BSPlib barrier performance prevents it being a viable (portable) alternative to SHMEM.

MPI-2

MPI-2 put/get is patterned on BSP, but with hooks that allow optimization for global memory machines (including non-global syncronization) [9]. If well implemented, this will provide a portable alternative to SHMEM. MPI-2 includes all of MPI-1, and it also includes a very powerfull parallel I/O interface. Thus parts of MPI-2 are usefull even for message passing codes. It is also possible to use MPI-1 message passing for some things and MPI-2 put/get for others. However, there are currently very few MPI-2 implementations (none from US vendors). Like BSPlib, MPI-2 uses memory windows and non-blocking puts and gets. However, MPI-2's Fortran interface is much superior to that in BSPlib. As is typical of MPI, the MPI-2 one-sided interface is very rich. It is as easy to write a Bulk Synchronous Parallel program with MPI-2 as with BSPlib, but this involves using a very particular small subset of MPI-2's one-sided capabilities. It does not seem easy to "emulate" SHMEM using MPI-2, and such an emulation would certainly not be portable to all machines that might benefit from put/get. Fortunately, translating a SHMEM program to use (portable) MPI-2 should be straight forward. However, the performance of MPI-2 global barriers will be critical if it is to replace SHMEM. Some of the non-global syncronization options in MPI-2 may improve performance over global barriers, but fast global barriers are going to be essential if MPI-2 is going to gain wide acceptance by SHMEM programmers.

F--

F-- is a simple extension to Fortran that allows SHMEM-like put/get to be expressed via assign statements [11]. At a minimum this is a much clearer way to express put/get than a subroutine call. There are more concrete advantages, including lower latency (no subroutine call overhead) and the possibility of applying all the usual compiler optimizations to remote memory accesses. As a language F-- is currently incomplete because it cannot conform to Fortran I/O semantics but does not provide an alternative. There are experimental versions of F-- for the SGI Origin 2000 and the Cray T3E, but no compilers from other vendors. A major potential advantage of F-- over SHMEM (or MPI-2) is compiler optimization of fine grain code fragments involving remote memory accesses. However, this has yet to be demonstrated in practice. One problem area for optimization is that the compiler must assume that any variable marked for remote access could in fact be remotely accessed at any time during execution of that subroutine (variables only need be marked in subroutines that perform remote access). This has the effect of drastically reducing the optimization possibilites for such variables, so F-- could end up being slower than the equivalent SHMEM (or OpenMP) code. This could have been avoided by providing a more relaxed memory model as part of the F-- definition.

OpenMP

OpenMP is a set of compiler directives that provide a high level interface to threads in Fortran, with both thread-local and global memory [12]. OpenMP can also be used for loop-level directive based parallelization, but in SPMD-mode N threads are spawned as soon as the program starts and exist for the duration of the run. The threads act like processes (e.g. in MPI), except that some memory is shared and there is a single I/O name space. There are alternatives, but the closest mapping to process-based SPMD programs is for almost all memory to be thread-local (i.e. one independent copy per thread) with global memory (visible to all threads) being used only as "buffers" for communication. A global buffer would typically hold N "local" buffers (one per thread). It is possible to use threads directly to create a threaded SPMD Fortran program, and portability is achievable via the Posix thread standard [13]. However, Posix threads are very low level and are difficult to use from Fortran. OpenMP provides a higher level, Fortran friendly, portable interface to threads. A threaded program has a single I/O space, and simultaneous calls from multiple threads may be unsafe. OpenMP has a more relaxed memory model than F--, that should not hinder optimization of shared variables.

HPF

High Performance Fortran provides a single-thread global memory user interface by doing communication and work distribution in the compiler, but it requires directives to distribute arrays across each processor's "local" memory [4].

Programming Issues

Portability

A language or library is "portable" if there are well understood guidelines for how to use (a subset of) the language or library so as to obtain good efficiency on a wide range machines (for a significant class of problems). SHMEM, F-- and OpenMP are unlikely to perform well on machines without a hardware global shared memory. BSPlib and MPI-2 put/get can take advantage of a hardware global shared memory, but can in principle also work on "shared nothing" systems, such as the IBM SP. How well MPI-2 will in fact work on such systems is unknown at present. A very low latency interconnect (and perhaps hardware support for barriers) might be all that is required to make MPI-2 put/get viable. Both HPF and MPI-1 can in principle be implemented efficiently on any scalable system.

MPI-1 is now available for all scalable systems, often via a vendor supported library. It is typically now possible to write a "portable" implementation of a given algorithm in MPI by following a few simple guidelines (defer syncronization, ISEND before IRECV, persistent communication requests, stride-1 buffers, don't use most collective operations). In addition, the syntax of MPI is regular enough that it is easy to provide several alternatives (selected at compile or run time). However, collective operations are often implemented very poorly. Thus a version using explicit point to point communications is almost always required for efficiency on some machines, with perhaps a MPI collective alternative for those few vendors who have optimized versions. Note that running many MPI collective operations twice on the same data is not guarenteed to produce the same result. This rules out such operations for many portable programs.

MPI-2 will probably become almost as widely available as MPI-1. It is not at all clear today what will be required to write portable put/get code using MPI-2. The key unanswered question is how easy will machines, such as the HP/Convex SPP-2000, with two kinds of memory (local and global) be to program using MPI-2 put/get. A secondary portability concern is how efficiently vendors implement the various syncronization options. Since the efficiency of MPI-2 put/get may be low on at least some "shared nothing" systems, programs that must run on such machines would have to at least provide a MPI-1 message passing alternative to each put/get. This reduces the ease of use advantage for put/get over message passing. It is relatively easy to add MPI-2 put/get as an option to an existing MPI-1 message passing program (selectively replacing only those operations that are faster using put/get).

BSPlib is available as source code for many machine types and there is an effort underway to get vendor's to produce optimized versions. However, given that BSPlib is quite slow on machines with a global shared memory and MPI-2 can be used to write BSP programs, there does not seem to be much future for BSPlib as a portability tool.

HPF is widely available, but the language standard was not designed for portability. For example, there are no portable default array distributions so a portable program must include compiler directives in every subroutine defining the layout of every array used by that subroutine. It is also still the case that alternative distributions can produce huge differences in performance and (more importantly) that different distributions perform well with different compilers. One approach to HPF portability is to use the Portland Group HPF compiler, which is available on many platforms (i.e. use a portable compiler, rather than a portable source code).

SHMEM is a very small library providing very fast put/get. However, no vendor other than SGI/Cray has chosen to provide an implementation. A portable program that uses SHMEM today must provide an alternative (typically MPI-1) for non-SGI machines. For those looking to migrate SHMEM programs to an API that is portable across shared memory machines, the viable options seem to be MPI-2 and OpenMP. MPI-2 provides put/get but with significant differences to SHMEM and with unknown performance. OpenMP is available today with performance comparable to SHMEM, but migrating from SHMEM to OpenMP may require changes to subroutines that don't currently call SHMEM. The issue of I/O is particularly problematic.

There are experimental versions of F-- for the SGI Origin 2000 and the Cray T3E, but no compilers from other vendors. If other compilers existed, the major portability issue would be performance which at least initially might be relatively low because of the memory model required for global variables. How to implement F-- on machines with both local and global memory would also be an issue. F-- has the best syntax of all the alternatives for SPMD Fortran on global shared memory machines, but without a portable (source to source) compiler or support from several major vendors it is not a viable portability tool.

OpenMP is available in beta today from SGI on the Origin 2000, and from KAI as a source to source compiler on several machine types. It has wide support and should soon be available on all machines with a global shared memory, from PC's to MPP's. The standard is not rigerous enough to be confident about portability between the many compilers that will exist. For example, it does not define the memory type (SHARED vs LOCAL) of variables with the SAVE atribute inside a subroutine. A program will definately break if a compiler allocates one kind of memory and the program assumes the other, so the only portable solution at present is to never use a SAVE statement in an OpenMP program (except for named COMMON). Once several implementations of OpenMP are available, it is likely that a portable subset of the language wil emerge. The only portable performance issue seems to be where global variables are placed in memory. OpenMP provides no mechanism to control this, and vendors are free to add their own (incompatable) extensions to OpenMP for laying out such arrays in memory. Some machines don't care about layout (e.g. Sun E10000) and some have run time layout mechanisms (e.g. SGI Origin 2000), but the performance on others may depend critically on shared array layout. Note that thread-local and shared variables map naturally to local and global memory respectively on machines with two kinds of memory. The only issue is where in global memory shared arrays are located.

Ease of Use

How easy each of the programming models is to use is obviously highly subjective. Message passing is certainly more difficult than put or get in that both sides of each memory transaction must cooperate in the exchange. This is more of an issue in cases with irregular communication patterns. The regular patterns typically associated with finite difference OCGM's are not usually difficult to express via message passing. The difficult part of put/get programming is syncronization, which is similar in all put/get models, but F-- is probably the easiest of all the process-based pure SPMD programming models to use.

A strong ease of use argument can be made for the global view of arrays provided by HPF. However, this is somewaht counter balanced by the difficulty of laying out arrays in memory. The extra boiler-plate code (compiler directives) needed for HPF programs is non-trivial. Many programmers seem to have "voted" for the less easy to use MPI-1, perhaps because HPF is easy to understand but does not necessarily provide a simple migration path from the existing code base. The performance of HPF relative to MPI-1 is also an issue.

OpenMP provides a programming model intermediate between F-- and HPF. It can use thread-local independent arrays, like F-- local arrays, or shared arrays, like HPF arrays, and can emulate F-- globally accessable local arrays using shared arrays with an extra dimension for the thread count. The primary difficulty with OpenMP is that SPMD threads that exist for the entire program are relatively new to Fortran programmers, and require some changes over process-based SPMD programming practices (particularly for I/O). Like all compiler directive based API's, the number of directives required can get out of hand (although it requires many fewer than HPF). OpenMP can be significantly easier to use than even F-- for irregular communications. For example a generic transpose operation in OpenMP might copy from one set of thread-local arrays (the input layout) into a shared array that uses the "natural" dimensioning and then copy out into a second set of thread-local arrays (the output layout). Both copy operations are trivial to program, and this works for any local distributions of the array. The real issue for OpenMP is not ease of use, but performance. In the transpose example, we have certainly done one extra copy of the entire array but this does not necessarily mean that this method is twice as slow as a direct copy from one layout to the other. In general, the fact that the programmer has no control over the layout of shared arrays in global memory might slow down some codes. However, threads are generally a big win over processes - particularly when mapping multiple threads or processes onto fewer processors.

I/O

Fortran has a specific model of I/O that is intrinsically single-thread, and which is violated by parallel I/O to a single file in all programming models except HPF. HPF can do parallel I/O that conforms to standard Fortran, but only if the compiler does this for you. All other API's except MPI-2 largely or completely ignore I/O. Generally serial writes from a single processor (or a single thread) works, as does parallel reads from any number of processors (but not from multiple threads). In some cases, parallel writes to non-overlapping records in a single file can be faster than serializing all writes - but there are no guarentees that this will work.

OpenMP has additional problems because there is just one process, and therefore one set of I/O files and pointers. Threaded I/O is actually well understood in C [13]. If the OpenMP Fortran's I/O library is "thread safe", any attempt to read and write in parallel to the same file (and perhaps to different files) will automatically be serialized. If the library is not safe, then the program must serialize I/O explicitly. Since there is only one I/O name space, only one thread should open and close a file and multiple reads of the same file from different threads will provide a different record to each thread. In contrast, for SPMD processes, each process must open and close a file it does I/O to and multiple reads of the same file from different processes will provide each with the same record.

NLOM inputs scalar control variables by reading them independently on all processors. This works well for process-based SPMD models, and is much less (programming) effort than the alternative of reading them on one processor and then broadcasting them to all others. This does not work for OpenMP, so NLOM now reads scalars into shared temporary variables from one thread under OpenMP (and into local temporary variables on all processes otherwise) and then copies the temporary variables into local variables on all threads/processes. This works with both threads and processes, but is not very transparent code. If OpenMP was the only target, it might be possible to leave input scalars in shared variables which would make the I/O code very similar to the uni-processor original (except for a few compiler directives).

MPI-2 contains an extensive API for parallel I/O. It is perhaps the most important reason for migrating from MPI-1 to MPI-2, particularly since the performance of MPI-2 put/get is as yet unknown. MPI-2 I/O looks like collective non-blocking message passing. Very general patterns of I/O are allowed, but probably a much smaller subset will actually provide good performance. Portability is an issue, particularly since the API includes potentially machine specific "hints" on file layout etcetera.

The fact that MPI-2 I/O is non-blocking implies that it is asynchronous I/O. On typical scalable systems, with huge memory capacities, it is often practical to buffer an entire dump of all prognostic variables. Which suggests that most OGCM's really require asynchronous I/O more than they do parallel I/O. There is no standard method for specifying asynchronous I/O in Fortran, but if it is available OpenMP can easily implement asynchronous array I/O using a shared memory buffer (even though parallel I/O is not typically possible). Similarly a HPF compiler might provide non-standard asynchronous I/O. The other programming models may need sufficient unused memory on a single processor (rather than globally) to hold an entire dump of all prognostic variables before asynchronous I/O becomes a possibility.

Computation and Communication

In the interests of portability and flexibility, NLOM (like many other domain decomposition codes) separates computation and communication into distinct phases of the algorithm (and into distinct subroutines). However, there are cases where overlap of computation and communication is desirable or even essential. BSPlib and SHMEM do not allow such overlap at all. MPI-2 put/get is non-blocking, but may be implemented like BSPlib on some machines. There are MPI-1 non-blocking message passing calls, which certainly reduce overall latency when sending several messages but may not allow true overlap of communication and computation. In HPF, all communciation is scheduled by the compiler and overlap of communication and computation is one way for the compiler to achieve good performance but it is largely outside the programmers control. F-- does not allow overlap except at the level of the compiler's scheduling of loads and stores, but it does provide very low latency which may make algorithms with intermixed communication and computation viable (also true to a lesser extent for SHMEM and MPI-2 put/get). OpenMP has similar latency to F--, and threads provide the only guaranteed user-level method to control the overlap of communication and computation (one thread communicates while another computes). OpenMP SPMD threads are not the most suitable starting point for this kind of thread use, but they are probably still easier to use than native threads. A good example of latency hiding by using threads is SC-MICOM [14], which hides the communication cost between SMP "boxes" by having more sub-tiles than processors and doing the sub-tiles near the edge of the tile first and then updating, via MPI-1, the halos with the other SMP boxes while the interior sub-tiles are calculated. This is also an example of two level parallelization (threads and MPI-1), which is probably going to become more common. The combination of OpenMP and MPI-1 provides the most opertunity for latency hiding, but MPI-2 put/get for near communication and and MPI-1 message passing for far communication is probably also going to become very common.

Porting to NLOM

NLOM was originally designed so that the single source code worked for data parallel compilers (CM Fortran) and for SPMD message passing. In addition to replacing 2-D loops with 4-D loops (which can also help in cache reuse), this required 2,600 HPF DISTRIBUTE directives and 500 HPF INDEPENDENT directives. The directives are implemented via cpp macros, to allow for machine and compiler specific variations (e.g. CM Fortran and HPF). The total code is 69,000 lines of Fortran 77 including 22,000 standard comment lines of which 500 are compiler directives (many are repeats in different dialects). In addition there are another 60,000 lines of comments in a standard format required for all Navy operational models. The communication API consists of 32 subroutines, and 10,000 lines of code are used in total to implement the various versions (autotasking, data parallel, MPI-1, SHMEM). There are 6,500 lines of code in five versions of 16 machine specific (primarily I/O) routines, and there is also significant parallel programming model specific, and machine specific, code in the direct Helmholtz's equation solver. Overall the single node version of NLOM would actually use 41,000 lines of code including 15,000 comments.

Adding support for OpenMP required generating 6,500 lines of code for OpenMP alone, although most of these are identical to the SHMEM version. The shared parts of the code required 900 OpenMP compiler directives, 500 to characterize all COMMON's (could be reduced using INCLUDE) and 400 primarily to handle I/O. The I/O logic required other modifications, as outlined in the I/O section above, so that all I/O is performed by the master thread only. The OpenMP standard does not allow SAVE to be used for local variables in a portable program. NLOM already used named COMMON for most such variables, because of previous portability problems with local SAVE. However, local variables initialized with a DATA statement are implicitly saved and several of these had to be removed from NLOM to allow OpenMP to work.

Adding MPI-2 put/get will formally require modifications to 6,500 lines of code, but most of these will be identical to the SHMEM version. Only 110 SHMEM GET calls will need replacing, plus any necessay modifications to the synchronization logic. Additional macros will be required to allow some subroutines to use MPI-1 and others to use MPI-2 on a machine by machine basis.

Since NLOM already has an array I/O API that is called collectively by all nodes (9 subroutines, 700 non-comment lines), adding MPI-2 I/O should be straight forward. For example, adding support for the IBM "Parallel I/O File System" required only 50 additional lines of code.

Test problems

Three NLOM-based benchmarks are used to evaluate performance. Source code is available at **ftp://ftp7320.nrlssc.navy.mil/pub/wobnch**.

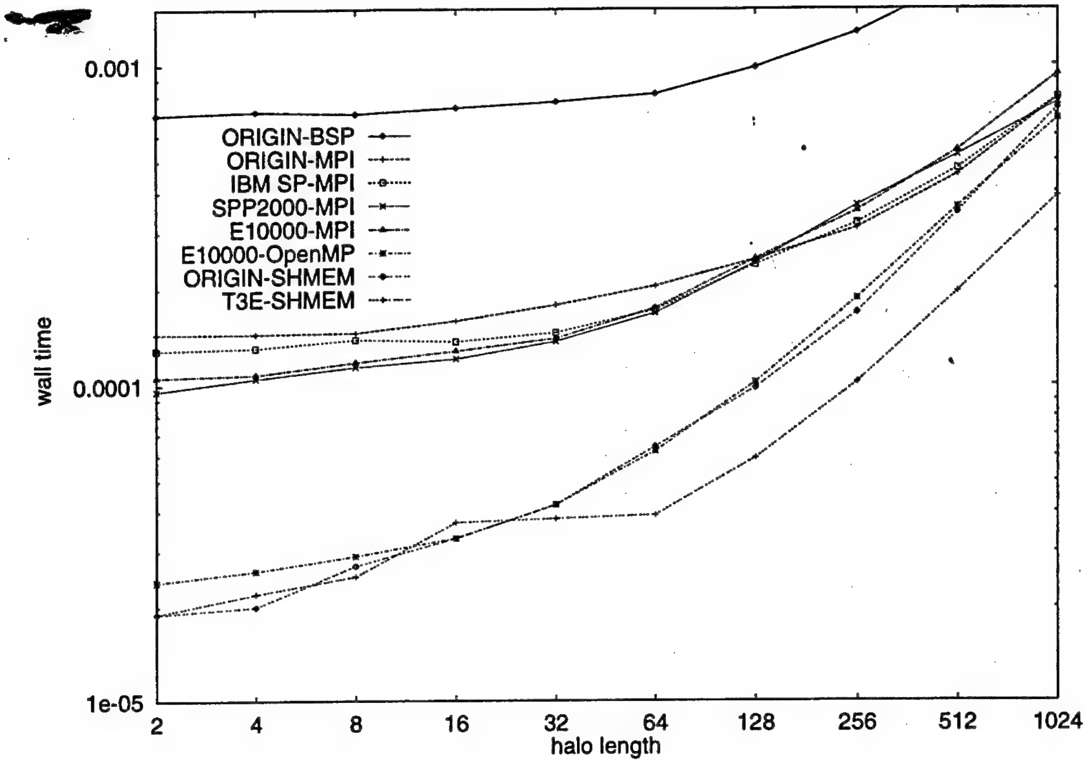

Figure 1: Best HALO times on 16 processors

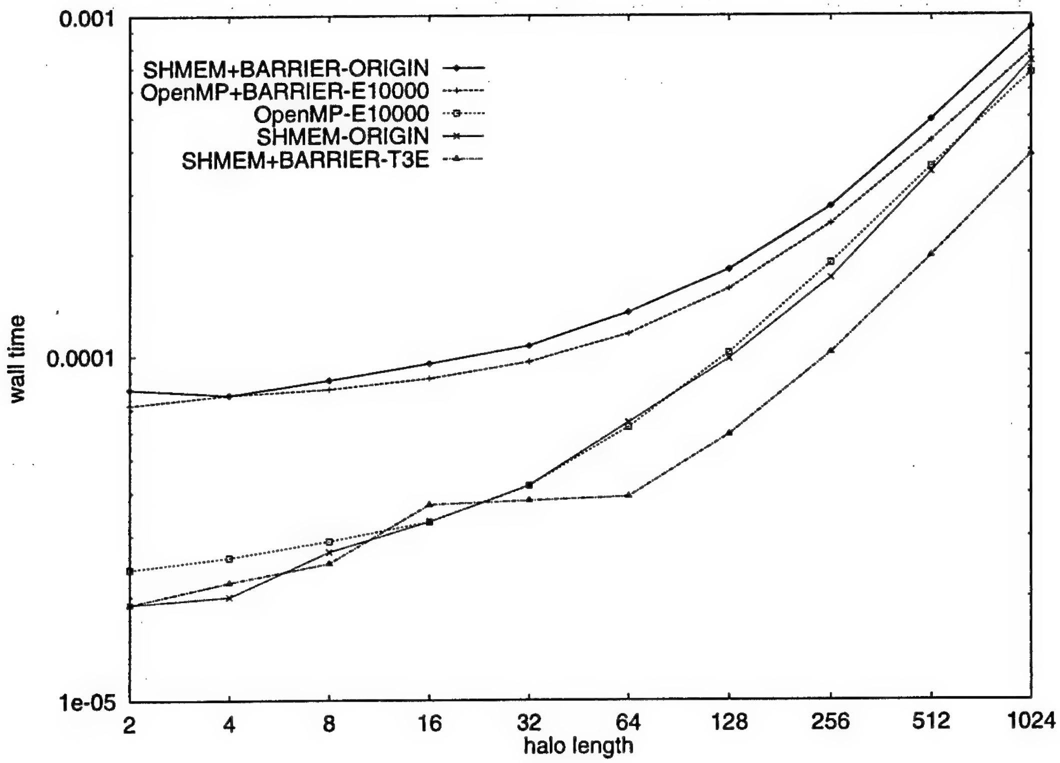

Figure 2: Shared memory HALO times on 16 processors

HALO

The HALO benchmark simulates a NLOM 2-D "halo" exchange for a N by N subdomain with $N = 2...1024$. There are separate versions for each programming model. These can be used to compare exchange strategies for a given programming model, or to intercompare models. HALO puts a premium on low latency, but so does NLOM as a whole and HALO performance correlates well with overall NLOM communication performance. Figure 1 shows performance for the best HALO implementation of several programming model on a range of 16-processor machines. BSPlib is very slow, apparently because a "superstep" barrier involves three actual barriers. The best MPI-1 implementation is typically persistent ISEND then IRECV, and MPI-1 performance is similar on all scalable systems shown. Note that the "shared nothing" IBM SP does about as well as shared memory systems using MPI-1. Finally, the 1-sided memory methods are fastest (i.e. have the lowest latency) where applicable. Figure 2 shows 1-sided memory methods in more detail, and illustrates that local synchronization is faster than global barriers except on the Cray T3E.

RBSOR

machine	library	nodes	RBSOR	XCTILR	XCNORM	speedup
Cray T3E	SHMEM	16	4.902	0.100	0.782	(450 MHz)
Cray T3E	SHMEM	32	2.035	0.077	0.414	2.41 x16
Cray T3E	SHMEM	64	1.115	0.067	0.233	1.83 x32
Cray T3E	SHMEM	128	0.580	0.046	0.123	1.92 x64
SGI Origin 2000	SHMEM	16	3.908	0.969	0.769	(195 MHz)
SGI Origin 2000	SHMEM	28	1.687	0.308	0.366	2.32 x16
SGI Origin 2000	SHMEM	56	0.924	0.199	0.218	1.83 x28
SGI Origin 2000	OpenMP	16	2.697	0.156	0.549	(195 MHz)
SGI Origin 2000	OpenMP	28	1.540	0.109	0.477	1.75 x16
SGI Origin 2000	OpenMP	56	1.061	0.285	0.299	1.45 x28
Sun E10000	Sun MPI	16	8.940	1.883	1.489	(250 MHz)
Sun E10000	Sun MPI	32	3.873	1.166	0.915	2.31 x16
Sun E10000	Sun MPI	56	1.793	0.504	0.501	2.16 x32
HP SPP-2000	HP MPI	16	3.401	0.486	0.651	(180 MHz)
HP SPP-2000	HP MPI	32	1.614	0.212	0.356	2.11 x16
HP SPP-2000	HP MPI	64	0.761	0.153	0.214	2.12 x32
IBM SP	IBM MPI	16	2.580	0.227	0.625	(135 MHz)
IBM SP	IBM MPI	32	1.562	0.204	0.465	1.65 x16
IBM SP	IBM MPI	64	0.955	0.163	0.324	1.64 x32
IBM SP	IBM MPI	128	0.892	0.167	0.411	0.98 x64

Table 1: Time in seconds for 27 2048x1344 Red-Black SOR solves

The RBSOR benchmark is a stand alone test of the red-black SOR iterative solver used by NLOM. Three wall clock times are recorded, a) total (RBSOR), b) halo exchange (XCTILR), and c) global sum (XCNORM). This benchmark is much simpler to get running than the full NLOM code, and it provides some indication of both computation and communication perforamance on a given machine. However, the computational kernal of RBSOR is not necessarily representative of NLOM as a whole (compare table 1, RBSOR, with table 2, NA824). The OpenMP times on a SGI Origin 2000 compare favorably with SHMEM times. The Sun E10000 is showing super-scalar speedup, but relatively poor computational kernal speed.

NA824

machine	method	nodes	time	Mflop/s	speedup
Cray T3E-900	SHMEM	14	44.1 mins	1,064	(450 MHz)
Cray T3E-900	SHMEM	28	21.0 mins	2,236	2.10x 14 nodes
Cray T3E-900	SHMEM	56	10.2 mins	4,591	2.06x 28 nodes
Cray T3E-900	SHMEM	112	5.7 mins	8,184	1.79x 56 nodes
Cray T3E-900	SHMEM	224	3.4 mins	13,601	1.68x112 nodes
SGI Origin 2000	SHMEM	14	75.3 mins	622	(195 MHz)
SGI Origin 2000	SHMEM	28	31.7 mins	1,481	2.38x 14 nodes
SGI Origin 2000	SHMEM	56	15.5 mins	3,031	2.05x 28 nodes
SGI Origin 2000	SHMEM	112	7.8 mins	6,030	1.99x 56 nodes
SGI Origin 2000	OpenMP	14	96.9 mins	484	(195 MHz)
SGI Origin 2000	OpenMP	28	38.0 mins	1,233	2.55x 14 nodes
SGI Origin 2000	OpenMP	56	21.1 mins	2,225	1.80x 28 nodes
SGI Origin 2000	OpenMP	112	12.7 mins	3,682	1.65x 56 nodes
HP SPP-2000	MPI	14	56.3 mins	833	(180 MHz)
HP SPP-2000	MPI	28	25.1 mins	1,868	2.24x 14 nodes
HP SPP-2000	MPI	56	15.1 mins	3,107	1.66x 28 nodes
IBM SP	MPI	14	39.2 mins	1,197	(135 MHz)
IBM SP	MPI	28	20.0 mins	2,345	1.96x 14 nodes
IBM SP	MPI	56	11.2 mins	4,169	1.79x 28 nodes
IBM SP	MPI	112	7.7 mins	6,060	1.45x 56 nodes
IBM SP	MPI	224	5.1 mins	9,208	1.51x112 nodes

Table 2: Performance of NLOM (NA824)

The NA824 benchmark is for 3.05 model days on a 1/32 degree 5-layer Atlantic Subtropical Gyre region (grid size 2048 x 1344 x 5). The run includes all the typical I/O and data sampling, but it does not measure initialization time (before the first time step). The sustained Mflops estimate is based on a hardware trace of a single processor Origin 2000 run (without MADD ops), i.e. is "useful" flops only. Like most heavily used benchmarks, this is for a problem smaller than those now typically run. The NA824 speedup from 28 to 56 processors is similar to the 112 to 224 speedup for the four times larger 1/64 degree Atlantic model. Illustrating that NLOM is indeed a "scalable" code. Table 2 summarizes the performance results. Note that for 28 processors and above 1/8th of the tiles are being discarded at compile time because they are over land, thus the 28 processor wall time is equivalant to a 32 processor wall time with no discarded tiles. Linear speedup from 14 to 28 processors is not 28/14 but 32/14 (i.e. not 2x but 2.29x). The Cray T3E is showing the best scalability to large numbers of nodes, but the IBM SP is competative on up to 64 processors. The SGI Origin 2000 is showing a sustained cache effect, with speedups of two or more for each doubling of nodes. OpenMP on the Origin is currently slower than SHMEM, but communication routines perform similarly between the two methods. So OpenMP compilation is slowing down the compuational kernals. This is a beta compiler and improvements can be expected in the future. The HP/Convex SPP-2000 is faster then the SGI Origin 2000 if only about half of the 16 processors in each hypernode are used (the 14 and 28 processor runs were on 2 and 4 hypernodes respectively). Like many other SMP systems, the SPP-2000's memory bandwidth does not sustain all the supplied processors when running memory bound jobs.

Conclusions

Retrofitting a scalable programming model to an existing scalable ocean code such as NLOM is not an ideal basis for comparision, even though NLOM is designed to accept alternative programming models. The separation of communication and computation phases for much of NLOM, and the fitting of each programming model into the existing communication API, puts at a disadvantage programming models that are easy to use and that favor mixing of communication and computation. Even so, this comparison provides a baseline for performance on an actual application. Early OpenMP compilers are showing promise, but MPI-2 put/get will probably be most programmers first exposure to 1-sided communication. We must hope that MPI-2 implementations will approach the performance of SHMEM and OpenMP.

Acknowledgements

This is a contribution to the 6.2 Global Ocean Prediction System Modeling Task. Sponsored by the Office of Naval Research under Program Element 62435N. Also to the Common HPC Software Support Initiative project Scalable Ocean Models with Domain Decomposition and Parallel Model Components. Sponsored by the DoD High Performance Computing Modernization Office. The benchmark simulations were performed under the Department of Defense High Performance Computing initiative, on (i) a SGI Origin 2000 and a HP SPP-2000 at the Naval Research Laboratory, Washington D.C., (ii) a Cray T3E at the Naval Oceanographic Office, Stennis Space Center, Mississippi, and (iii) an IBM SP at Waterways Experiment Station, Vicksburg, Mississippi.

References

[1] *Hurlburt, H.E. & J.D. Thompson (1980). A numerical study of Loop Current intrusions and eddy shedding.* J.Phys. Oceanogr., **10**, *pp 1611-1651.*

[2] *Wallcraft A.J (1991). The NRL Layered Ocean Model users guide. NOARL Report* **35**. *Naval Research Laboratory, Stennis Space Center, MS, 21 pp.* **http://www7300.nrlssc.navy.mil/html/images/users_guide.ps.gz**

[3] *Moore D.R. & A.J. Wallcraft (1996). Formulation of the NRL Layered Ocean Model in Spherical Coordinates. NRL Contractor Report* **CR 7323–96-0006** *Naval Research Laboratory, Stennis Space Center, MS.*

[4] *Koelbel C.H., D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., M.E. Zosel (1994). The High Performance Fortran handbook. MIT Press.*

[5] *Wallcraft A.J & D.R. Moore (1996). A Scalable Implementation of the NRL Layered Ocean Model. NRL Contractor Report* **CR 7323–96-0005** *Naval Research Laboratory, Stennis Space Center, MS.*

[6] *Wallcraft A.J & D.R. Moore (1997). The NRL Layered Ocean Model.* Parallel Computing **23**, *pp 2227-2242.*

[7] *Snir M., S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra (1996). MPI: the complete reference. MIT Press.*

[8] *Cray Research Inc. (1996) Application Programmer's Library Reference Manual. Cray Research SR-2165*

[9] *Message Passing Interface Forum (1997) MPI-2: Extensions to the Message Passing Interface.* **http://www.mpi-forum.org/docs/docs.html**

[10] *Goudreau, M.W., J.M.D. Hill, K. Lang, B. McColl, S.B. Rao, D.C. Stefanescu, T. Suel, T. Tsantilas (1996) A proposal for the BSP worldwide standard library.* **http://www.bsp-worldwide.org**

[11] *Numrich R.W., J.L. Steidel, B.H. Johnson, B.D, de Dinechin, G. Elsesser, G. Fischer, T. MacGonald (1997) Definition of the* F-- *extension to Fortran 90. Proc. 10th Int. Workshop on Language and Compilers for Parallel Computers Springer-Verlag*

[12] *OpenMP Organization (1997) OpenMP Fortran Application Programming Interface* **http://www.openmp.org**

[13] *Kleiman, S., D. Shah, B. Smaalders (1996) Programming with threads SunSoft Press. Prentice Hall.*

[14] *A. Sawdey, M. O'Keefe, W. Jones. A General Programming Model for Developing Scalable Ocean Circulation Applications. 1996 ECMWF Workshop on the Use of Parallel Processors in Meteorology* **http://www-mount.ee.umn.edu/ okeefe/micom/**